JINJIE Segment Tree

线段树进阶

Jul.8, 2024

陳旻庚

目 录

MU LU

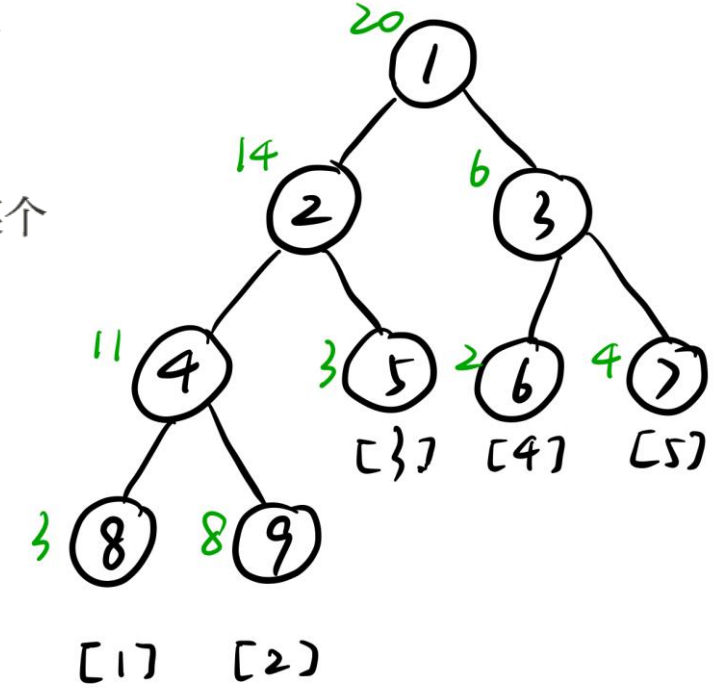
复习

线段树 (Segment Tree) 用于维护满足 **结合律** 的区间信息，支持在 $\Theta(\log n)$ 的时间内 **区间修改** 和 **区间查询**。

线段树是一棵 **平衡二叉树**，树上每个 **节点 p** 都存储一条 **线段（区间）** $[l, r]$ 的某个性质，节点 p 的左右子节点的编号分别为 $2p$ 和 $2p+1$，分别储存 $[l, \text{mid}]$ 和 $[\text{mid}+1, r]$。

修改时，使用 **懒标记 mark_p**，表示该节点 p 对应的区间上每一个点都要加上某个数，用到它的 **子区间** 的时候，再向下 **传递** 修改。

预处理时间复杂度 $\Theta(n)$，查询或修改时间复杂度 $\Theta(\log n)$。



复习

如何写线段树？

- 抄板子：下传、回溯、建树、查询、更新
- 维护什么？push_up()
- 如何修改？push_down()

```cpp
struct SEGTREE {
    ll val; // 区间和
    ll mrk; // 懒标记
} tree[N << 2];
#define ls p<<1
#define rs p<<1|1
int n, arr[N];

// 在这两个函数中写需要维护的东西
inline void push_up(SEGTREE& p, SEGTREE l, SEGTREE r) {
    p.val = l.val + r.val;
}
inline void push_down(SEGTREE& p, int mrk, int len) {
    p.val += mrk * len;
    p.mrk += mrk;
}

// 下传与回溯
inline void push_up(int p) { push_up(tree[p], tree[ls], tree[rs]); }
inline void push_down(int p, int len) {
    if (len <= 1) return;                                   // 传递到树的叶 结束
    push_down(tree[ls], tree[p].mrk, len - len / 2);        // 左子区间传递
    push_down(tree[rs], tree[p].mrk, len / 2);              // 右子区间传递
    tree[p].mrk = 0;                                        // 传递后 标记清空
}

// 建树 当前节点区间[cl,cr] 节点编号p
void build(int p = 1, int cl = 1, int cr = n) {
    tree[p].mrk = 0;                                        // 初始化
    if (cl == cr) return tree[p].val = arr[cl], void();     // 到树的叶 结束
    int mid = (cl + cr) >> 1;
    build(ls, cl, mid);                                     // 左子区间[l,mid]
    build(rs, mid + 1, cr);                                 // 右子区间[mid+1,r]
    push_up(p);                                             // 更新当前节点信息
}

// 查询[l,r] 当前节点区间[cl,cr] 节点编号p
SEGTREE query(int l, int r, int p = 1, int cl = 1, int cr = n) {
    if (cl >= l && cr <= r) return tree[p];  // 查询的区间包含当前节点区间 返回当前节点区间
    push_down(p, cr - cl + 1);
    int mid = (cl + cr) >> 1;
    if (mid >= r) return query(l, r, ls, cl, mid);
    if (mid < l)  return query(l, r, rs, mid + 1, cr);
    SEGTREE ans;
    push_up(ans, query(l, r, ls, cl, mid), query(l, r, rs, mid + 1, cr));
    return ans;
}

// 为[l,r]每个数加d 当前节点区间[cl,cr] 节点编号p
void update(int l, int r, ll d, int p = 1, int cl = 1, int cr = n) {
    if (cl >= l && cr <= r) return push_down(tree[p], d, cr - cl + 1);  // 查询的区间包含当前节点区间
    push_down(p, cr - cl + 1);
    int mid = (cl + cr) >> 1;
    if (mid >= l) update(l, r, d, ls, cl, mid);
    if (mid < r)  update(l, r, d, rs, mid + 1, cr);
    push_up(p);    // 更新当前节点信息
}
```

复习

Tips：

- 用结构体存储
- 查询函数返回结构体
- 能封装为函数的就写函数

```
struct SEGTREE {
    ll val; // 区间和
    ll mrk; // 懒标记
} tree[N << 2];
#define ls p<<1
#define rs p<<1|1
int n, arr[N];

// 在这两个函数中写需要维护的东西
inline void push_up(SEGTREE& p, SEGTREE l, SEGTREE r) {
    p.val = l.val + r.val;
}
inline void push_down(SEGTREE& p, int mrk, int len) {
    p.val += mrk * len;
    p.mrk += mrk;
}

// 下传与回溯
inline void push_up(int p) { push_up(tree[p], tree[ls], tree[rs]); }
inline void push_down(int p, int len) {
    if (len <= 1) return;                                   // 传递到树的叶 结束
    push_down(tree[ls], tree[p].mrk, len - len / 2);        // 左子区间传递
    push_down(tree[rs], tree[p].mrk, len / 2);              // 右子区间传递
    tree[p].mrk = 0;                                        // 传递后 标记清空
}

// 建树 当前节点区间[cl,cr] 节点编号p
void build(int p = 1, int cl = 1, int cr = n) {
    tree[p].mrk = 0;                                        // 初始化
    if (cl == cr) return tree[p].val = arr[cl], void();     // 到树的叶 结束
    int mid = (cl + cr) >> 1;
    build(ls, cl, mid);                                     // 左子区间[l,mid]
    build(rs, mid + 1, cr);                                 // 右子区间[mid+1,r]
    push_up(p);                                             // 更新当前节点信息
}

// 查询[l,r] 当前节点区间[cl,cr] 节点编号p
SEGTREE query(int l, int r, int p = 1, int cl = 1, int cr = n) {
    if (cl >= l && cr <= r) return tree[p];  // 查询的区间包含当前节点区间 返回当前节点区间
    push_down(p, cr - cl + 1);
    int mid = (cl + cr) >> 1;
    if (mid >= r) return query(l, r, ls, cl, mid);
    if (mid < l)  return query(l, r, rs, mid + 1, cr);
    SEGTREE ans;
    push_up(ans, query(l, r, ls, cl, mid), query(l, r, rs, mid + 1, cr));
    return ans;
}

// 为[l,r]每个数加d 当前节点区间[cl,cr] 节点编号p
void update(int l, int r, ll d, int p = 1, int cl = 1, int cr = n) {
    if (cl >= l && cr <= r) return push_down(tree[p], d, cr - cl + 1);  // 查询的区间包含当前节点区间
    push_down(p, cr - cl + 1);
    int mid = (cl + cr) >> 1;
    if (mid >= l) update(l, r, d, ls, cl, mid);
    if (mid < r)  update(l, r, d, rs, mid + 1, cr);
    push_up(p);    // 更新当前节点信息
}
```

复习

回忆：

- [x] 维护区间和(E)、区间最值(A)、区间异或和（或、与）……
- [x] 单点修改、区间加(E)、区间乘(FG)、
  区间异或(G)、区间赋值(今天讲)……

*：括号内字母是 Spring-Training Round #15 中的题号

复习

C. 教室里有 n 盏灯，从左到右依次编号为：$1, 2, \ldots, n$，初始时都是开着的。人走灯灭，下课后，m 个同学依次经过灯的开关，每个人会执行以下两种操作中的一种：

- 操作 1：关上从 l 到 r（含边界）范围内的所有灯；
- 操作 2：打开从 l 到 r（含边界）范围内的所有灯。

请你帮助班长计算，为了关上所有的灯，他还需要关上几盏灯。

维护什么?

如何修改?

复习

C. 区间赋值：

```
21  // 在这两个函数中写需要维护的东西
22  inline void push_up(SEGTREE& p, SEGTREE l, SEGTREE r) {
23      p.val = l.val + r.val;
24  }
25  inline void push_down(SEGTREE& p, int mrk, int len) {
26      p.val = mrk * len;
27      p.mrk = mrk;
28  }
29
30  // 下传与回溯
31  inline void push_up(int p) { push_up(tree[p], tree[ls], tree[rs]); }
32  inline void push_down(int p, int len) {
33      if (len <= 1 || tree[p].mrk == -1) return;          // 这里没有标记 返回
34      push_down(tree[ls], tree[p].mrk, len - len / 2);    // 左子区间传递
35      push_down(tree[rs], tree[p].mrk, len / 2);          // 右子区间传递
36      tree[p].mrk = -1;                                   // 传递后 标记清空
37  }
```

复习

C.

Constraints

$1 \leqslant n \leqslant 10^9, \quad 1 \leqslant m \leqslant 3 \cdot 10^5, \quad 1 \leqslant l_i, r_i \leqslant n, \quad 1 \leqslant k_i \leqslant 2$

Runtime error on test 5

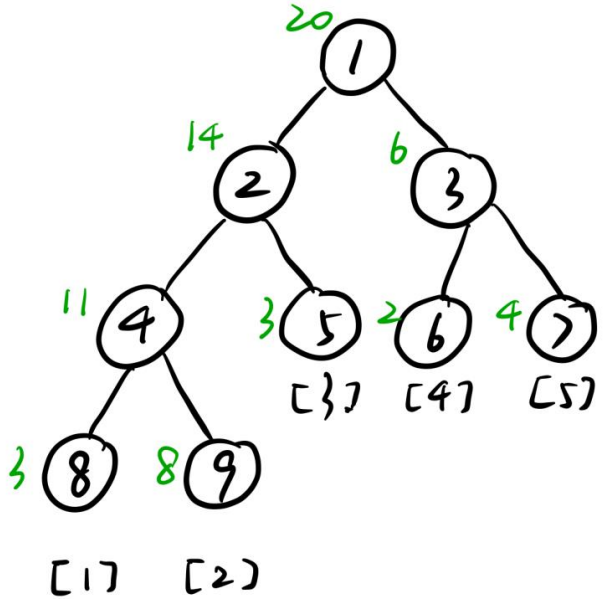
PART 01

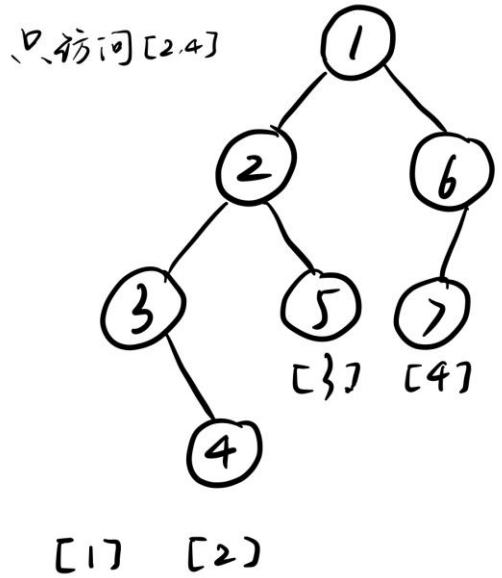
动态开点

DONG TAI KAI DIAN

动态开点

当值域为 v 时，普通的线段树的空间复杂度是 $\Theta(4v)$。如果 v 巨大，但查询次数 q 较小时，边修改边建树，这样空间的复杂度是 $\Theta(q \log v)$。





动态开点

动态开点的核心在于「开」，即用哪个节点，就新建哪个节点

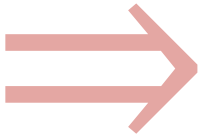
```
inline void push_down(int p, ll mrk, int len) {
    tree[p].val += mrk * len;
    tree[p].mrk += mrk;
}
```

```
inline void push_down(int& p, ll mrk, int len) {
    if (!p) p = ++pcnt;
    tree[p].val += mrk * len;
    tree[p].mrk += mrk;
}
```

动态开点

节点与其左右儿子不再满足简单的函数关系，故用两个变量来存储：

```
const int N = 3e5 + 8;
struct SEGTREE {
    ll val; // 区间和
    ll mrk; // 懒标记
} tree[N << 2];
#define ls p<<1
#define rs p<<1|1
```



```
const int N = 3e7 + 8;
struct SEGTREE {
    ll val; // 区间和
    ll mrk; // 懒标记
    int ls, rs; // 左右儿子
} tree[N];
```

数组能开多大就开多大，避免 RE

在调用左右儿子时也需要相应地修改：

```
// 为[l,r]每个数加d 当前节点区间[cl,cr] 节点编号p
void update(int l, int r, ll d, int p = 1, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return push_down(p, d, cr - cl + 1);
    push_down(p, cr - cl + 1);
    if (mid >= l) update(l, r, d, ls, cl, mid);
    if (mid < r)  update(l, r, d, rs, mid + 1, cr);
    push_up(p);
}
```

```
void update(int l, int r, ll d, int p = 1, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return push_down(p, d, cr - cl + 1);
    push_down(p, cr - cl + 1);
    if (mid >= l) update(l, r, d, tree[p].ls, cl, mid);
    if (mid < r)  update(l, r, d, tree[p].rs, mid + 1, cr);
    push_up(p);
}
```

递归的起点是区间 $[L,R]$，依数据范围修改，允许是负数。注意，这里的 `>>1` 不能写为 `/2` 。

```
int main() {
    int n = read();
    L = 1, R = n, pcnt = 0;  // 指定递归起点为初始化数据范围[L,R] 初始化节点时间戳

    L = 0, R = 1e9, pcnt = 0;

    L = -1e5, R = 1e5, pcnt = 0;
```

动态开点

C. 完整代码：

```cpp
#include <cstdio>
using ll = long long;
const int N = 16000000 + 8;

inline ll read() {
    ll Y = getchar(), S = 0, C = 1;
    while (Y > 57 || Y < 48) { if (Y == 45) C = -1; Y = getchar(); }
    while (Y <= 57 && Y >= 48) { S = (S << 3) + (S << 1) + Y - 48; Y = getchar(); }
    return S * C;
}

struct SEGTREE {
    int val, mrk = -1;
    int ls, rs;
} tree[N];
int L, R, pcnt;
#define mid (cl+cr>>1)

// 在这两个函数中写需要维护的东西
inline void push_up(SEGTREE& p, SEGTREE l, SEGTREE r) {
    p.val = l.val + r.val;
}
inline void push_down(int& p, int x, int len) {
    if (!p) p = ++pcnt;     // 开点
    tree[p].val = x * len;
    tree[p].mrk = x;
}

// 下传与回溯
inline void push_up(int p) { push_up(tree[p], tree[tree[p].ls], tree[tree[p].rs]); }
inline void push_down(int p, int len) {
    if (len <= 1 || tree[p].mrk == -1) return;          // 这里没有标记 返回
    push_down(tree[p].ls, tree[p].mrk, len - len / 2);  // 左子区间传递
    push_down(tree[p].rs, tree[p].mrk, len / 2);        // 右子区间传递
    tree[p].mrk = -1;                                   // 传递后 标记清空
}

```

```
SEGTREE query(int l, int r, int p = 1, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return tree[p];  // 查询的区间包含当前节点区间 返回当前节点区间
    push_down(p, cr - cl + 1);
    if (mid >= r) return query(l, r, tree[p].ls, cl, mid);
    if (mid < l)  return query(l, r, tree[p].rs, mid + 1, cr);
    SEGTREE ans;
    push_up(ans, query(l, r, tree[p].rs, mid + 1, cr), query(l, r, tree[p].ls, cl, mid));
    return ans;
}

void update(int l, int r, int d, int p = 1, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return push_down(p, d, cr - cl + 1);
    push_down(p, cr - cl + 1);
    if (mid >= l) update(l, r, d, tree[p].ls, cl, mid);
    if (mid < r)  update(l, r, d, tree[p].rs, mid + 1, cr);
    push_up(p);
}

int main() {
    int n = read();
    L = 1, R = n, pcnt = 1;  // 初始化数据范围[L,R]
    update(1, n, 1);
    for (int q = read(); q--; ) {
        int l = read(), r = read(), op = read();
        update(l, r, op - 1);
        printf("%d\n", query(1, n).val);
    }
    return 0;
}
```

PART 02

线段树二分

XIAN DUAN SHU ER FEN

线段树二分

例：非负序列，单点修改，全局查询前缀和大于 S 的第一个位置 p，$n \leqslant 5 \times 10^5$, $q \leqslant 5 \times 10^6$。

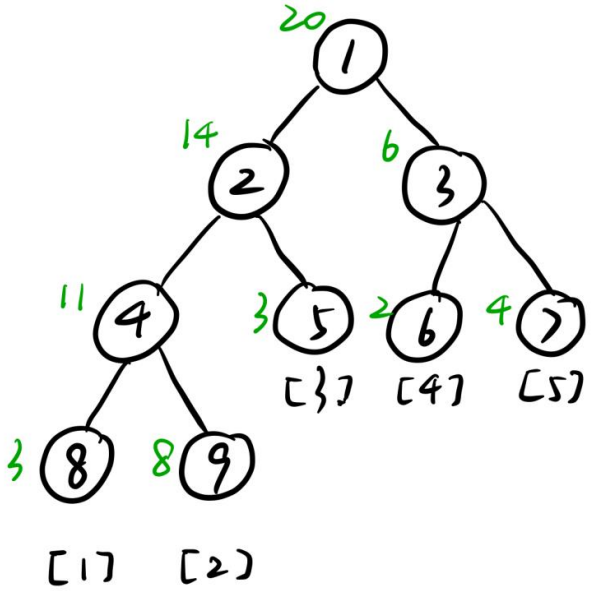
线段树二分

例：非负序列，单点修改，全局查询前缀和大于 S 的第一个位置 p，$n \leqslant 5 \times 10^5$，$q \leqslant 5 \times 10^6$。

线段树维护区间和。对于朴素二分，每次 check 需查询单点前缀和，一次询问复杂度 $\Theta(\log^2 n)$；而线段树的分治结构 **合并二分和查询的过程**，实现 $\Theta(\log n)$ 的复杂度。

考察二分过程，第一次查询 $[1,n]$ 中点 m 处的前缀和 s_m，它等于线段树根节点左儿子的权值。若 $s_m > S$，说明答案在 $[1,m]$，否则答案在 $[m+1,n]$。对于前者，第二次查询 $[1,m]$ 中点 m_l 处的前缀和 s_{m_l}，它等于根节点左儿子的左儿子的权值；对于后者，第二次查询 $[m+1,r]$ 中点 m_r 处的前缀和，它等于 s_m 加上根节点右儿子的左儿子的权值……不断递归，直至进入叶子节点。

以 cur 记录已查询到的累计前缀和。递归至节点 p 时，查询左儿子的权值 val_{p_l}，比较 $\mathrm{cur} + \mathrm{val}_{p_l}$ 与 S 的大小，如果 $\mathrm{cur} + \mathrm{val}_{p_l} > S$，向左递归，cur 不变；否则向右递归，并 $\mathrm{cur} := \mathrm{cur} + \mathrm{val}_{p_l}$。



线段树二分

以 cur 记录已查询到的累计前缀和。递归至节点 p 时，查询左儿子的权值 val_{p_l}，比较 $\mathrm{cur}+\mathrm{val}_{p_l}$ 与 S 的大小，如果 $\mathrm{cur}+\mathrm{val}_{p_l}>S$，向左递归，cur 不变；否则向右递归，并 $\mathrm{cur}:=\mathrm{cur}+\mathrm{val}_{p_l}$。

```
// 全局二分 已查询到的累计前缀和&cur 限制lim 当前节点区间[cl,cr] 节点编号p
int binary(ll& cur, ll lim, int p = 1, int cl = 1, int cr = n) {
    if (cur + tree[p].val <= lim) return cur += tree[p].val, -1; // 全加上都不够 那目标位置不在当前区间 全加上然后返回-1
    if (cl == cr) return cl;                                     // 到叶子 既然能过的来那一定是目标
    push_down(p);
    int res = binary(cur, lim, ls, cl, mid);     // 先向左递归找
    if (res != -1) return res;                   // 左边不是-1说明左边有解 那不用管右边了
    return binary(cur, lim, rs, mid + 1, cr);    // 左边无解再看右边 右边无论有没有解都是看右边
}
```

线段树二分

在区间 $[l,r]$ 查询前缀和大于 S 的第一个位置 p，大同小异，直接给出代码：

```
// 区间[l,r]局部二分 已查询到的累计前缀和&cur 限制lim 当前节点区间[cl,cr] 节点编号p
int binary(int l, int r, ll& cur, ll lim, int p = 1, int cl = 1, int cr = n) {
    if (cl >= l && cr <= r) {  // 查询区间包含当前区间
        if (cur + tree[p].val <= lim) return cur += tree[p].val, -1; // 全加上都不够 那目标位置不在当前区间 全加上然后返回-1
        if (cl == cr) return cl;                                     // 到叶子 是目标
        // 否则继续二分
    }
    push_down(p);
    if (mid >= l) {
        int res = binary(l, r, cur, lim, ls, cl, mid);               // 先向左递归找
        if (res != -1) return res;                                   // 左边有解
    }
    if (mid < r) return binary(l, r, cur, lim, rs, mid + 1, cr);     // 左边无解再看右边
    return -1;
}
```

具有单调性才能二分！

线段树二分灵活性很强，没有绝对固定的模板

复习

树状数组求逆序对

```
6
5 4 2 6 3 1
```

```
11
```

```cpp
int main() {
    int n = read();
    for (int i = 1; i <= n; ++i) {
        a[i] = read();
    }

    ll ans = 0;
    for (int i = n; i >= 1; --i) {
        ans += query(a[i]);
        update(a[i], 1);
    }
    printf("%lld\n", ans);

    return 0;
}
```

思考：局限性?

权值树状数组 —— 权值线段树

用线段树维护 **桶**，在 $\Theta(\log v)$ 的时间内查询某个范围内的数出现的总次数、第 k 大数、前驱后继等。空间复杂度 $\Theta(q \log v)$。

PART 03

权值线段树

QUAN ZHI XIAN DUAN SHU

权值线段树

用线段树维护 **桶**，在 $\Theta(\log v)$ 的时间内查询某个范围内的数出现的总次数、第 k 大数、前驱后继等。空间复杂度 $\Theta(q \log v)$。

权值线段树

在动态开点线段树代码里补充：

```
// 插入
void insert(int x) {
    update(x, x, 1);
}

// 删除
void remove(int x) {
    update(x, x, -1);
}
```

权值线段树

在动态开点线段树代码里补充：

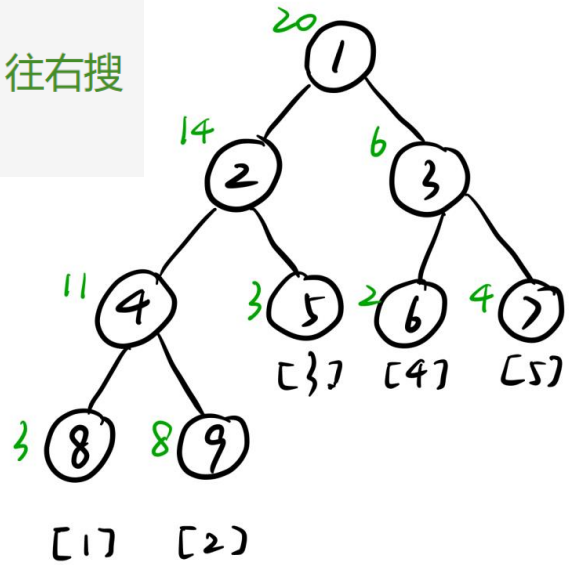
```
// 小于x的数量
int countl(int x) {
    return query(L, x - 1).val;
}

// 大于x的数量
int countg(int x) {
    return query(x + 1, R).val;
}
```

权值线段树

在动态开点线段树代码里补充：

```
int kthnum(int x, int p = rt, int cl = L, int cr = R) {    // 指定排名的数
    if (cl == cr) return cl;
    if (tree[tree[p].ls].val >= x)
        return kthnum(x, tree[p].ls, cl, mid);     // 往左搜
    else
        return kthnum(x - tree[tree[p].ls].val, tree[p].rs, mid + 1, cr);    // 往右搜
}
```



权值线段树

在动态开点线段树代码里补充：

```
// 前驱predecessor
int prenum(int x) {
    return kthnum(countl(x));
}

// 后继successor
int sucnum(int x) {
    return kthnum(tree[1].val - countg(x) + 1);
}
```

权值线段树

用线段树维护 **桶**，在 $\Theta(\log v)$ 的时间内查询某个范围内的数出现的总次数、第 k 大数、前驱后继等。空间复杂度 $\Theta(q \log v)$。

权值线段树无需懒标记 `mrk`，由于是单点更新，也无需 `push_down` 和 `push_up` 函数，而是递归到底、区间相加，代码相对于普通线段树更短，这也是较 **平衡树** 的优势，但是当值域较大、询问较多时空间占用较大。

精简后的代码：

```
14  struct SEGTREE {
15      int val, ls, rs;
16  } tree[N];
17  int L, R, rt, pcnt;  // 递归起点 根节点 节点时间戳
18
19  void update(int x, int d, int& p = rt, int cl = L, int cr = R) {
20      if (!p) p = ++pcnt;
21      if (cl == cr) return tree[p].val += d, void();
22      int mid = (cl + cr) >> 1;
23      if (x <= mid) update(x, d, tree[p].ls, cl, mid);
24      if (x > mid)  update(x, d, tree[p].rs, mid + 1, cr);
25      tree[p].val = tree[tree[p].ls].val + tree[tree[p].rs].val;
26  }
27
28  int query(int l, int r, int p = 1, int cl = L, int cr = R) {
29      if (cl >= l && cr <= r) return tree[p].val;
30      int mid = (cl + cr) >> 1;
31      if (mid >= r) return query(l, r, tree[p].ls, cl, mid);
32      if (mid < l)  return query(l, r, tree[p].rs, mid + 1, cr);
33      return query(l, r, tree[p].rs, mid + 1, cr) + query(l, r, tree[p].ls, cl, mid);
34  }
35
36  inline void insert(int x) { update(x, 1); }                // 插入
37  inline void remove(int x) { update(x, -1); }               // 删除
38  inline int countl(int x) { return query(L, x - 1); }
39  inline int countg(int x) { return query(x + 1, R); }
40  inline int rankof(int x) { return countl(x) + 1; }         // 排名 (比当前数小的数的个数+1)
41  int kthnum(int x, int p = rt, int cl = L, int cr = R) {    // 指定排名的数
42      if (cl == cr) return cl;
43      int mid = (cl + cr) >> 1;
44      if (tree[tree[p].ls].val >= x) return kthnum(x, tree[p].ls, cl, mid);     // 往左搜
45      else return kthnum(x - tree[tree[p].ls].val, tree[p].rs, mid + 1, cr);    // 往右搜
46  }
47  inline int prenum(int x) { return kthnum(countl(x)); }  // 前驱
48  inline int sucnum(int x) { return kthnum(tree[rt].val - countg(x) + 1); }  // 后继
```

PART 04

线段树合并(一)

XIAN DUAN SHU HE BING

线段树合并

Description

请用某种数据结构维护一个图，您的代码应当能够实现：

1. 新建一个节点，权值为 a；
2. 连接两个节点 a, b；
3. 将一个节点 a 所属于的联通块内权值小于 b 的所有节点权值变成 b；
4. 将一个节点 a 所属于的联通块内权值大于 b 的所有节点权值变成 b；
5. 询问一个节点 a 所属于的联通块内的第 b 小的权值是多少；

更具象化的描述：

永无乡包含 n 座岛，编号从 1 到 n，每座岛都有自己的独一无二的重要度，按照重要度可以将这 n 座岛排名，名次用 1 到 n 来表示。某些岛之间由巨大的桥连接，通过桥可以从一个岛到达另一个岛。如果从岛 a 出发经过若干座（含 0 座）桥可以 到达岛 b，则称岛 a 和岛 b 是连通的。

现在有两种操作：

`B x y` 表示在岛 x 与岛 y 之间修建一座新桥。

`Q x k` 表示询问当前与岛 x 连通的所有岛中第 k 重要的是哪座岛，即所有与岛 x 连通的岛中重要度排名第 k 小的岛是哪座，请你输出那个岛的编号。

线段树合并

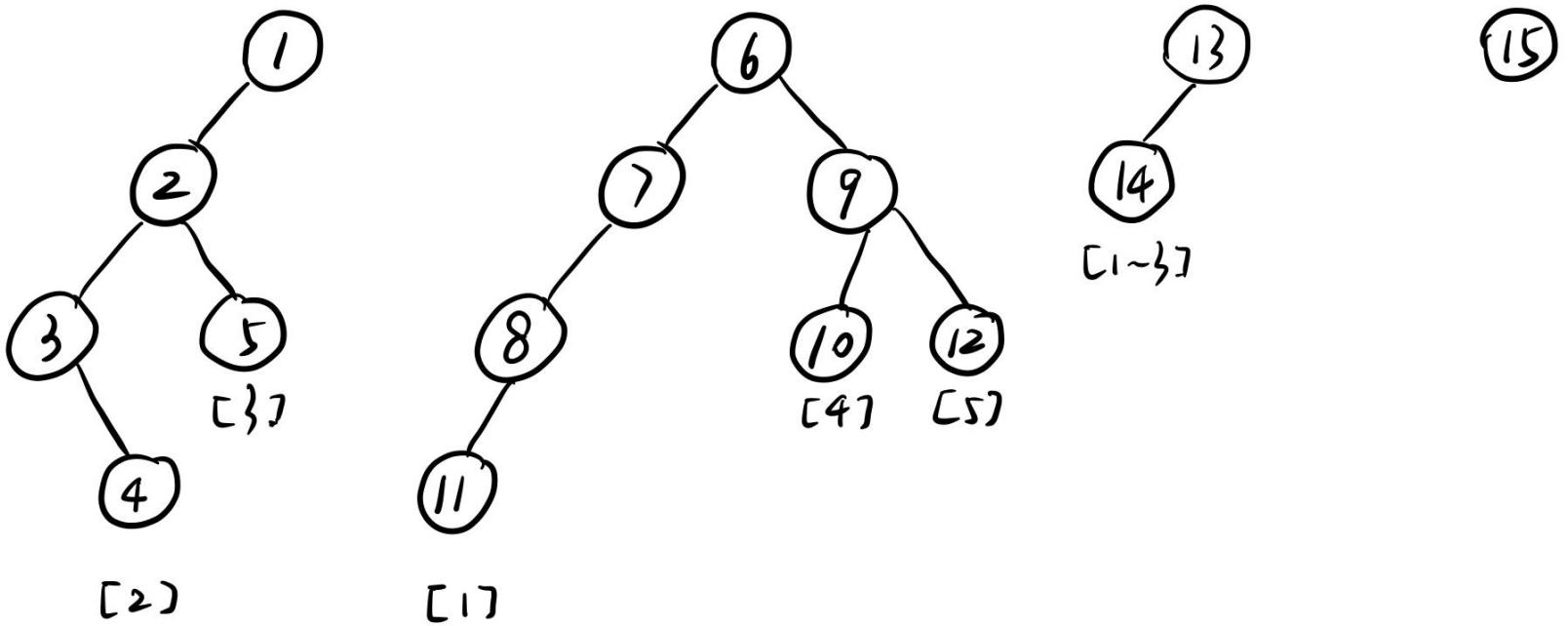
用并查集维护连通性。为了查询每个连通块里的第 k 大，自然想到权值线段树，每个联通块都需要维护一个权值线段树。在并查集合并联通块的同时合并线段树。

如何建 n 棵树?

如何「合并」权值线段树?

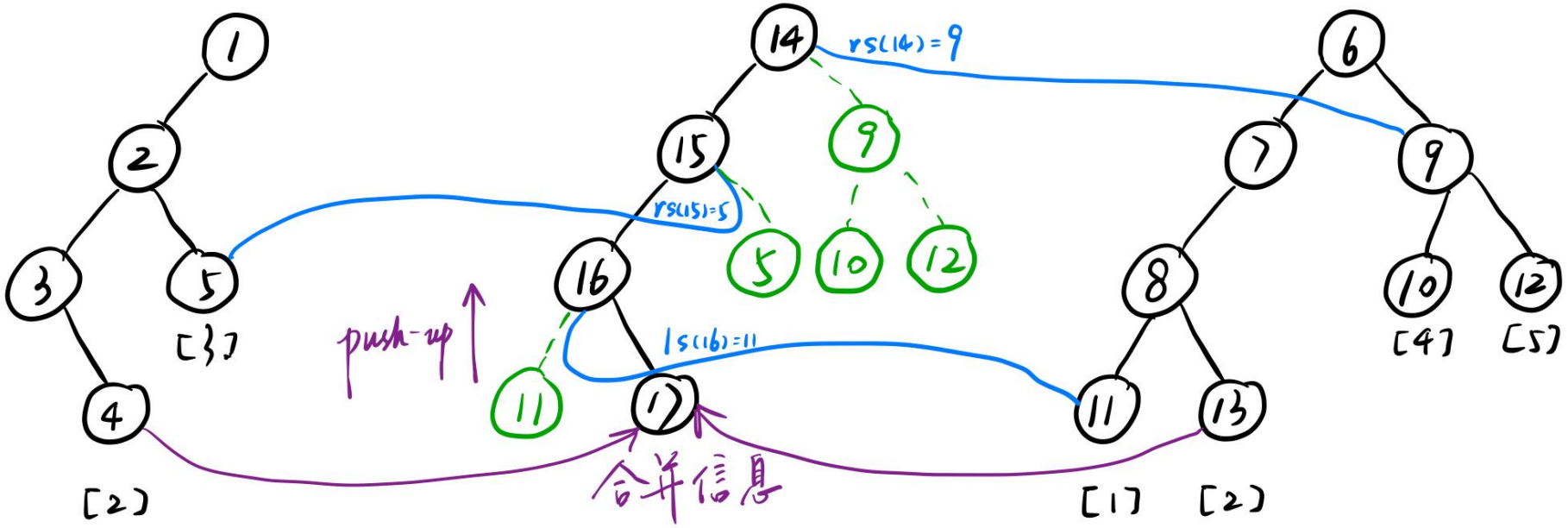
线段树合并

如何建 n 棵树?



线段树合并

如何合并权值线段树?



线段树合并

动态开点， rt[p] 表示节点 p 所在树的根节点 unix。递归地将两棵线段树的节点信息对应合并：

```
// 将树上节点x,y合并到节点z
int merge(int x, int y, int cl = L, int cr = R) {
    if (!x || !y) return x | y;                    // 一方为空 递归到叶 返回非空的那个节点
    int z = ++pcnt;
    if (cl == cr) return /* 合并叶子节点x,y 将y的信息传递给x */ z;
    tree[z].ls = merge(tree[x].ls, tree[y].ls, cl, mid);       // x的左节点和y的左节点合并为z的左节点
    tree[z].rs = merge(tree[x].rs, tree[y].rs, mid + 1, cr);   // x的右节点和y的右节点合并为z的右节点
    push_up(z);                                                // 更新当前节点信息
    return z;
}

void eachT() {
    L = 1, R = n, pcnt = 0;
    for (int i = 1; i <= n; ++i) {
        rt[i] = ++pcnt;            // 建立根节点
        update(num, 1, rt[i]);     // 将num插入权值线段树i 指定树的根是rt[i]
    }  // 建n颗权值线段树
    for (int q = read(); q--; ) {
        rt[x] = rt[y] = merge(rt[x], rt[y]);      // 从树根开始合并x,y 并将x和y的根设置为新根
    }
}
```

线段树合并

上面的写法保留原来两棵线段树的信息，创建了一棵新树，因此需要额外的一倍空间。如果不要求保留原来的信息，优先选择直接把第二棵树的信息加到第一棵树上，这不需要新建节点，但会毁掉原先的两棵树：

```
// 将树上节点x,y合并到节点x
int merge(int x, int y, int cl = L, int cr = R) {
    if (!x || !y) return x | y;       // 一方为空 递归到叶 返回非空的那个节点
    if (cl == cr) return /* 合并叶子节点x,y 将y的信息传递给x */ x;
    tree[x].ls = merge(tree[x].ls, tree[y].ls, cl, mid);       // x的左节点和y的左节点合并为x的左节点
    tree[x].rs = merge(tree[x].rs, tree[y].rs, mid + 1, cr);   // x的右节点和y的右节点合并为x的右节点
    push_up(x);                                                // 更新当前节点信息
    return x;
}
```

线段树合并

完整代码

```
/// ----------------线段树----------------
struct SEGTREE {
    int siz, mrk;
    double mul;
    int ls, rs;
} tree[M];
int  n, L, R, pcnt, rt[N];
#define mid (cl+cr>>1)

// 在这两个函数中写需要维护的东西
inline void push_up(SEGTREE& p, SEGTREE& l, SEGTREE& r) {
    p.siz = l.siz + r.siz;
    p.mul = l.mul + r.mul;
}
void push_down(SEGTREE& p) {
    p.siz = p.mul = 0;
    p.mrk = 1;
}

// 下传与回溯
inline void push_up(int& p) { push_up(tree[p], tree[tree[p].ls], tree[tree[p].rs]); }
void push_down(int& p) {
    if (!tree[p].mrk) return;
    push_down(tree[tree[p].ls]);
    push_down(tree[tree[p].rs]);
    tree[p].mrk = 0;
}

// 插入d个x 当前节点区间[cl,cr] 节点编号p
void update(int x, int d, int& p, int cl = L, int cr = R) {
    if (!p) p = ++pcnt;
    tree[p].siz += d, tree[p].mul += d * log2(x);
    if (cl == cr) return;
    push_down(p);
    if (x <= mid) update(x, d, tree[p].ls, cl, mid);
    else update(x, d, tree[p].rs, mid + 1, cr);
    push_up(p);
}

// [l,r]赋值为0
void update2(int l, int r, int& p, int cl = L, int cr = R) {
    if (!p) return;
    if (l <= cl && cr <= r) return push_down(tree[p]);
    push_down(p);
    if (l <= mid) update2(l, r, tree[p].ls, cl, mid);
    if (r > mid)  update2(l, r, tree[p].rs, mid + 1, cr);
    push_up(p);
}
```

```
// 查询[l,r] 当前节点区间[cl,cr] 节点编号p
int query(int l, int r, int& p, int cl = L, int cr = R) {
    if (!p) return 0;
    if (l <= cl && cr <= r) return tree[p].siz;
    push_down(p);
    int ans = 0;
    if (l <= mid) ans += query(l, r, tree[p].ls, cl, mid);
    if (r > mid)  ans += query(l, r, tree[p].rs, mid + 1, cr);
    return ans;
}

// 权值线段树模板
int kthnum(int k, int& p, int cl = L, int cr = R) {
    if (cl == cr) return cl;
    push_down(p);
    if (k <= tree[tree[p].ls].siz) return kthnum(k, tree[p].ls, cl, mid);
    else return kthnum(k - tree[tree[p].ls].siz, tree[p].rs, mid + 1, cr);
}

// 将树上节点x,y合并到节点x
int merge(int x, int y, int cl = L, int cr = R) {
    if (!x || !y) return x | y;      // 一方为空 递归到叶 返回非空的那个节点
    if (cl == cr) return push_up(tree[x], tree[x], tree[y]), x;
    push_down(x), push_down(y);
    tree[x].ls = merge(tree[x].ls, tree[y].ls, cl, mid);      // x的左节点和y的左节点合并为x的左节点
    tree[x].rs = merge(tree[x].rs, tree[y].rs, mid + 1, cr);  // x的右节点和y的右节点合并为x的右节点
    push_up(x);                                               // 更新当前节点信息
    return x;
}

/// ----------------并查集----------------
int fa[N];
void init() { for (int i = 1; i < N; ++i) fa[i] = i; }
int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }
void uno(int x, int y) { fa[y] = x; }
```

线段树合并

```
/// ---------------主函数---------------
int main() {
    L = 1, R = 1e9, pcnt = 0;
    init();  // 并查集要初始化
    for (int q = read(); q--; ) {
        int op = read(), a, b, num;
        switch (op) {
            case 1:
                // 新建
                a = read();
                update(a, 1, rt[++n]);  // 将1个a插入新节点的权值线段树
                break;
            case 2:
                // 连接
                a = find(read()), b = find(read());
                if (a == b) break;  // 不能合并自己
                rt[a] = rt[b] = merge(rt[a], rt[b]);
                uno(a, b);
                break;
            case 3:
                // 把小于b的变成b
                a = find(read()), b = read();
                num = query(L, b, rt[a]);  // 需要删除L~b所有的数 共num个
                update2(L, b, rt[a]);          // 删除L~b
                update(b, num, rt[a]);     // 将num个b插入权值线段树n
                break;
            case 4:
                // 把大于b的变成b
                a = find(read()), b = read();
                num = query(b, R, rt[a]);
                update2(b, R, rt[a]);
                update(b, num, rt[a]);
                break;
            case 5:
                // 求a的第b小
                a = find(read()), b = read();
                printf("%d\n", kthnum(b, rt[a]));
                break;
            case 6:
                // 比较a和b的乘积的大小
                a = find(read()), b = find(read());
                printf("%d\n", tree[rt[a]].mul > tree[rt[b]].mul);
                break;
            case 7:
                // 节点a的siz
                a = find(read());
                printf("%d\n", tree[rt[a]].siz);  // 根节点的siz 无需使用query函数
                break;
```

新建权值线段树

连通并查集的同时合并线段树

PART 05

线段树合并（二）

XIAN DUAN SHU HE BING

线段树合并

Description

有一个 n 个节点的森林[†]，节点编号为 1 到 n，森林中每棵树都是有根的。

如果 a 是 b 的父节点，那么称 a 为 b 的 1 级祖先；如果 b 有一个 1 级祖先，a 是 b 的 1 级祖先的 $k-1$ 级祖先，那么称 a 为 b 的 k 级祖先。

如果存在一个节点 z，是两个节点 a 和 b 共同的 p 级祖先：那么称 a 和 b 为 **p 级表亲**。

给出 m 次询问，每次询问给出一对整数 v 和 p，求编号为 v 的节点有多少个 p 级表亲。

线段树合并

某个节点的 p-级表亲的深度相同，每次询问即 v 的 p-级祖先的深度为 dep_p 的儿子个数。

树具有天然的合并性质，即某节点的子树信息全部来自于它子节点的子树信息。因此先计算子节点，再计算父节点，在树上 DFS 的过程中，将子树的信息合并到父节点上。

```
void dfs(int u) {
    rt[u] = ++pcnt;                          // 为这个节点建一棵树
    update(dep[u], 1, rt[u]);                // 把信息存进去
    for (auto v : E[u]) {
        dfs(v);                              // 先把子树的信息算好
        rt[u] = merge(rt[u], rt[v]);         // 把子树的信息存到这个节点上
    }
    // 计算与这个节点相关的答案
    //
}
```

线段树合并

回到本题，

维护什么？

权值线段树维护 **每一深度的节点个数**，区间加。

如何更新？

每遍历一个点，就向权值线段树中插入这个点的深度。

如何合并？

权值线段树合并模板，略。

如何查询？

先离线存储询问，对于询问 v, p，存储 p-级祖先和 dep_p（如果 p-级祖先不存在，则不存储，此时答案是 0）。这样只需要统计每一个节点 u 对应的深度 dep 的答案，即查询 u 对应的权值线段树上从 dep 到 dep 的数量，`query(dep, dep, rt[u])`。

线段树合并

提示：

1. 题给的图是森林，可以创建一个虚拟的「超级爸爸」是每个树根的祖先，将森林化为一棵树。（警钟敲烂！新增节点会导致深度 +1，也会导致小于某个深度的节点总数 +1，不理解的看 CTSC1997 选课）
2. 求 v 的 p-级祖先，类似于求 LCA，倍增向上跳：

```
std::vector<int> E[N];
int Log2[N], fa[33][N], dep[N], w[N];
int rt[N];

void initlca(int u, int fau = 0) {
    dep[u] = dep[fau] + 1;
    fa[0][u] = fau;
    for (int i = 1; i <= Log2[dep[u]]; ++i) {
        fa[i][u] = fa[i-1][fa[i-1][u]];
    }  // u的第2^{i-1}个父节点的第2^{i-1}个父节点即第2^{i}个父节点
    for (auto v : E[u]) {
        initlca(v, u);
    }
}

// v的k级祖先
```

```
int getast(int v, int k) {
    k = dep[v] - k;  // 期望的深度
    if (k <= 0) return -1;  // v没有k级祖先
    while (dep[v] > k)
        v = fa[Log2[dep[v] - k]][v];
    return v;
}
```

3. **离线存储答案：**

```
std::vector<std::pair<int, int> > que[N];
int ans[N];

int q = read();
for (int i = 1; i <= q; ++i) {
    int v = read(), p = read();
    int fav = getast(v, p);              // v的p级祖先
    que[fav].push_back({ dep[v],i });    // 存储每个询问 第二维是编号id
}

/* 计算答案 */

for (int i = 1; i <= q; ++i) {
    printf("%d ", ans[i]);
}
```

线段树合并

完整代码

```cpp
/// ----------------线段树----------------
struct SEGTREE {
    ll val;
    int ls, rs;
} tree[M];
int L, R, pcnt;  // 递归起点 节点时间戳
#define mid (cl+cr>>1)

inline void push_up(SEGTREE& p, SEGTREE l, SEGTREE r) { p.val = l.val + r.val; }
inline void push_up(int p) { push_up(tree[p], tree[tree[p].ls], tree[tree[p].rs]); }

int query(int l, int r, int p, int cl = L, int cr = R) {
    if (cl >= l && cr <= r) return tree[p].val;
    if (mid >= r) return query(l, r, tree[p].ls, cl, mid);
    if (mid < l)  return query(l, r, tree[p].rs, mid + 1, cr);
    return query(l, r, tree[p].rs, mid + 1, cr) + query(l, r, tree[p].ls, cl, mid);
}

void update(int x, int d, int& p, int cl = L, int cr = R) {
    if (!p) p = ++pcnt;
    if (cl == cr) return tree[p].val += d, void();
    if (x <= mid) update(x, d, tree[p].ls, cl, mid);
    if (x > mid)  update(x, d, tree[p].rs, mid + 1, cr);
    push_up(p);
}

// 将树上节点x,y合并到节点x
int merge(int x, int y, int cl = L, int cr = R) {
    if (!x || !y) return x | y;                                  // 一方为空 递归到叶 返回非空的那个节点
    if (cl == cr) return tree[x].val += tree[y].val/* 合并叶子结点x,y 将y的信息传递给x */, x;
    tree[x].ls = merge(tree[x].ls, tree[y].ls, cl, mid);         // x的左节点和y的左节点合并为x的左节点
    tree[x].rs = merge(tree[x].rs, tree[y].rs, mid + 1, cr);     // x的右节点和y的右节点合并为x的右节点
    push_up(x);                                                  // 更新当前节点信息
    return x;
}

/// ----------------树+LCA----------------
std::vector<int> E[N];
int Log2[N], fa[33][N], dep[N], w[N];
int rt[N];

void initlca(int u, int fau = 0) {
    dep[u] = dep[fau] + 1;
    fa[0][u] = fau;
    for (int i = 1; i <= Log2[dep[u]]; ++i) fa[i][u] = fa[i - 1][fa[i - 1][u]];
    for (auto v : E[u]) initlca(v, u);
}

// v的k级祖先
int getast(int v, int k) {
    k = dep[v] - k;  // 期望的深度
    if (k <= 1) return -1;  // v没有k级祖先
    while (dep[v] > k) v = fa[Log2[dep[v] - k]][v];
    return v;
}
```

```cpp
/// ----------------线段树合并----------------
std::vector<std::pair<int, int> > que[N];
int ans[N];

void dfs(int u) {
    rt[u] = ++pcnt;                         // 为这个节点建一棵树
    update(dep[u], 1, rt[u]);               // 把信息存进去
    for (auto v : E[u]) {
        dfs(v);                             // 先把子树的信息算好
        rt[u] = merge(rt[u], rt[v]);        // 把子树的信息存到这个节点上
    }
    // 计算与这个节点相关的答案
    for (auto [depth, id] : que[u]) {
        ans[id] = query(depth, depth, rt[u]) - 1;
    }
}

/// ----------------主函数----------------
int main() {
    int n = read();
    L = 0, R = n + 1, pcnt = 0;
    for (int i = 2; i < N; ++i) Log2[i] = Log2[i / 2] + 1;
    for (int i = 1; i <= n; ++i) {
        int x = read();
        if (!x) x = n + 1;  // 给森林一个虚拟根节点n+1
        E[x].push_back(i);
    }
    initlca(n + 1);
    int q = read();
    for (int i = 1; i <= q; ++i) {
        int v = read(), p = read();
        int fav = getast(v, p);                 // v的p级祖先
        if (fav == -1) ans[i] = 0;              // v没有p级祖先 答案是0
        else que[fav].push_back({ dep[v],i });  // 存储每个询问
    }

    dfs(n + 1);  // 线段树合并 计算答案

    for (int i = 1; i <= q; ++i) {
        printf("%d ", ans[i]);
    }

    return 0;
}
```

PART 05

杂例

ZA LI

杂例 – 线段树应用

关键词：连续序列、区间……

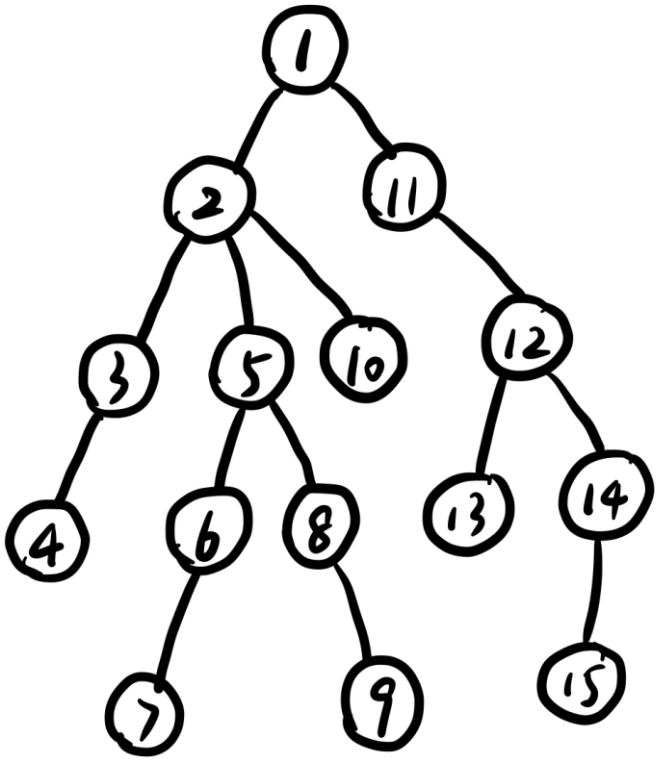
线段树合并在树上问题作用显著

杂例 – 线段树应用

回忆 **树的 DFS 序**：每个节点在 DFS 中的进出栈的时间序列。每棵子树的 DFS 序都是连续的，且子树上根节点 DFS 序最小。因此，u 的子树上所有节点的编号介于 `dfsn[u]` 和 `dfsnR[u]` 之间，其中 `dfsnR[u]` 是结束遍历 u 的子树的最后一个节点的 DFS 序。

这个良好性质 **将树转化为 (广义的) 区间**，对 u 的 **子树** 的操作即是对 **区间** `dfsn[u]` $\sim$ `dfsnR[u]` 的操作。

这支持在 $\Theta(\log n)$ 的时间内实现单点修改查询、子树修改查询、路径修改查询，在树根固定的问题中应用广泛。

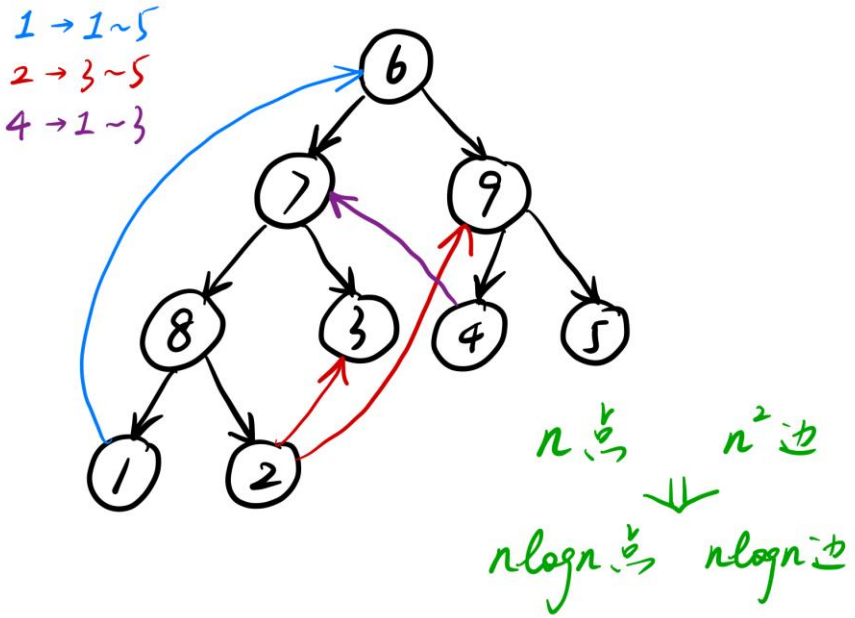


杂例 – 线段树应用

有 n 个点，q 个询问，每次询问给出一个操作，在这些操作后求 1 到 n 的最短路。$n \leqslant 1 \times 10^5$, $q \leqslant 1 \times 10^5$。

- `1 u v w`：连一条 $u \to v$ 的有向边，权值为 w；
- `2 u l r w`：对于所有 $i \in [l, r]$，连一条 $u \to i$ 的有向边，权值为 w；
- `3 u l r w`：对于所有 $i \in [l, r]$，连一条 $i \to u$ 的有向边，权值为 w；
- `4 l1 r1 l2 r2 w`：对于所有 $i \in [l_1, r_1], j \in [l_2, r_2]$，连一条 $i \to j$ 的有向边，权值为 w；

无论何种最短路方法，从建图就卡住了。注意到，题给建图是区间，联想到线段树，可以利用线段树减少连边数量，从而降低复杂度。



参考

1. 小明同學：数据结构 – ST表 树状数组 线段树 (https://kobicgend.top/posts/8303a602.html)
2. 小明同學：线段树进阶 (https://kobicgend.top/pdf/JINJIESegmentTree.pdf)
3. Alex_Wei：线段树进阶 Part 1 (https://www.luogu.com.cn/article/r1mp3hga)
4. Maoyiting：「算法笔记」线段树优化建图 (https://www.cnblogs.com/maoyiting/p/13764109.html)
5. feecle6418：线段树合并：从入门到精通 (https://www.luogu.com.cn/training/3858#problems)
6. ducati：提高组树上问题全面提升综合题单 (https://www.luogu.com.cn/training/34654#problems)
7. EnofTaiPeople：小清新线段树上二分 (https://www.luogu.com.cn/article/kcghapi0)
8. 逍遥Fau：Greedy Shopping(线段树) (https://blog.csdn.net/weixin_45799835/article/details/109777927)
9. CCSU_梅子酒：【2022ICPC沈阳I题解】【值域线段树+贪心】The 2022 ICPC Asia Shenyang Regional Contest I. Quartz Collection (https://blog.csdn.net/TT6899911/article/details/130536214)
10. xishuiw：2022CCPC女生赛-L.彩色的树（线段树合并）(https://www.cnblogs.com/xishuiw/p/18004799)
11. NaHCO3_tht：权值线段树 (https://www.luogu.com.cn/article/mf2fx8v5)
12. subarude：P5025【[SNOI2017]炸弹】(https://www.luogu.com.cn/article/62tq0m7k)
13. Krystallos：题解 P5025【[SNOI2017]炸弹】(https://www.luogu.com.cn/article/55ow5kwh)

ZHU All Killed YU KUAI

祝 A K 愉 快

Jul.8, 2024

陳旻庚